

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
APPLICATION FOR PATENT

INVENTOR: Tim Wilkinson and Peter Mehlitz

TITLE: Portable Operating Environment for Information Devices

ATTY DOCKET: TRAS-520

PRIORITY

This application claims the benefit of priority to United States provisional patent application no. 60/225,569, filed August 14, 2000.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to software architectures for networked computing devices, which are operated in heterogenous network environments and are able to adapt to varying content types.

2. Discussion of the Related Art

Today, most operating environments are still focused on general purpose desktop computers, and are tightly coupled to their underlying operating systems, a typical example being Microsoft Windows. For certain information device categories, there exist specialized environments like Epoch (cell phones) and Palm OS (Palm PDAs). Several technologies exist which can be utilized in different, cross-platform

environments (e.g. Personal Java VMs). These systems so far do not provide the desire degree of:

- portability (different HW architectures)
- scalability (different speed, memory, formfactors)
- integration (no redundant components, functions)
- extensibility (update/install of SW components via network to adapt to new content types)

for coping with today's rapidly changing device categories and content types. In particular, large device vendors are sought after for a common software infrastructure which is flexible enough to serve the needs of very different, special purpose, and usually resource-constrained devices.

Most existing drivers focus on expensive desktop-oriented windowing systems (Windows, X etc.), and accelerated display hardware. Consequently, these drivers have a high degree of runtime configurability (color depth, screen resolution), and use a design which assumes that most performance critical functions can be offloaded to special purpose hardware (bit block transfers, line drawing etc.). Inherently, such a design exhibits a considerable amount of indirection (functions called via pointers), which is acceptable if this overhead is compensated by fast graphics hardware. There are some adaptations of such drivers for generic framebuffer devices, but because of the display hardware oriented design, they do not make use of platform specific optimization mechanisms.

In general, applications use toolkit library functions, which in turn map to driver functions, either by means of direct calls or some inter-process communication mechanism. The driver then branches according to the display HW type (display processor), the user configured screen resolution (width, height) and color depth,

invoking the functions which actually do the work (mostly by delegating it to the display processor).

Embedded computing devices mostly don't have accelerated display HW, and provide just a framebuffer memory area. They usually have a fixed resolution and color depth, i.e. don't have a high degree of runtime configurability. They mostly have RISC like processors which provide efficient mechanisms for memory block transfers (number of registers). It is desired to have a graphics rendering mechanism for use with an embedded device which is designed to compensate for a lack of expensive display hardware.

To be suitable for global deployment, it is desired to be able to use local font sets. To overcome the problem of various different, and partially conflicting character encoding schemes, the Unicode initiative has become a de-facto standard. It is based on multi-byte character encoding and as of today contains about 50,000 character symbols.

This constitutes a considerable problem for displaying unicode text with conventional font rendering systems. In general, a font object includes of a collection of per-font and per-character attribute values, the character attributes being index-accessed (with the character code) for efficiency reasons.

The per-character attribute (width, bearing, glyph etc.) implementations usually use arrays of the size of the required character set. While this involves just about 10kB for a bitmapped 256-character font instance (e.g. ASCII), a full unicode font (50,000 characters) would use up to 2 MB for a single instance (face/weight/slant/size). Typical applications (e.g. web-browsers) utilize about 4 different instances, adding up to 8-10MB of required font space. This amount of memory is not generally available for embedded devices (16 MB RAM/ROM). It is

therefore desired to have a font extension mechanism which permits the implementation of a reasonable number of font instances in less than ,e.g., 2 MB of memory.

In order to provide a suitable basis for a broad range of applications for use with embedded devices, it is recognized herein that it would be advantageous to implement a full windowing system with z-order capabilities (i.e. windows can overlap, with foreground windows partially or fully obscuring background windows). In addition, the rendering mechanism should support explicitly set, non-z-order implied clipping rectangles. Devices having special display processors may implement this by means of HW-augmented region clipping. As mentioned above, embedded devices usually do not have special display processors. It is desired to provide an embedded device with a rendering mechanism that supports explicitly set, non-z-order implied clipping rectangles.

Handwriting recognition systems known in the art include the GRAFFITI system, developed by Xerox, which is used in the PalmPilot, and the JOT system, which is used in WinCE (and others). It is desired to have an improved handwriting recognition system, particularly for use with an embedded computing device.

SUMMARY OF THE INVENTION

A graphics rendering software program for use on an embedded computing device includes an application layer, a graphics toolkit, and a graphics driver. the graphics driver includes a shape function layer and a framebuffer access macro layer. The shape function layer includes a target architecture specific instruction set for setting and retrieving pixel values, respectively, into and from a one-dimensional framebuffer memory. The framebuffer access macro layer includes a set of macros for inlining into the shape function layer.

A method for rendering graphics on a display of an embedded computing device is further provided including setting and retrieving pixel values, respectively, into and from a one-dimensional framebuffer memory of a shape function layer of a graphics rendering software running on the embedded computing device, and inlining macros into the shape function layer.

A graphics driver of a graphics rendering software program for use with an embedded device is further provided including a shape function layer including a target architecture specific instruction set for setting and retrieving pixel values, respectively, into and from a one-dimensional framebuffer memory, and a framebuffer access macro layer including a set of macros for inlining into the shape function layer.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 schematically illustrates a software architecture according to a preferred embodiment.

Figure 2 schematically illustrates processes for retrieving content using a client support server according to a preferred embodiment.

Figure 3 schematically illustrates a process for retrieving simplified content converted from complex content by a converter service on a client support server according to a preferred embodiment.

Figure 4 schematically illustrates a process for retrieving a software package or update using a package manager module on a client and a package repository on a client support server according to a preferred embodiment.

Figure 5 schematically illustrates a process for retrieving a software package or update using a package manager module, local service registry and local service on

a client and a package repository on a client support server according to a preferred embodiment.

Figure 6 schematically illustrates a graphics rendering mechanism according to a preferred embodiment.

Figure 7 schematically illustrates scene cohesion of a graphics rendering mechanism according to a preferred embodiment.

Figure 8 schematically illustrates a scanline cell according to a preferred embodiment.

Figure 9 schematically illustrates different color depths.

Figure 10 schematically illustrates the unicode font character set.

Figure 11 schematically illustrates a font extension mechanism according to a preferred embodiment.

Figure 12 schematically illustrates a graphics rendering mechanism for providing overlapping drawings surfaces on an embedded device display according to a preferred embodiment.

Figure 13 schematically illustrates schematically illustrates a graphics rendering mechanism for providing overlapping drawings surfaces on an embedded device display using rectangular clip segments according to a preferred embodiment.

Figure 14 schematically illustrates different visible portions of an obscured drawing surface as sets of rectangular clipping segments.

Figure 15 schematically illustrates theming according to a preferred embodiment.

Figure 16 schematically further illustrates theming according to a preferred embodiment.

Figure 17 schematically further illustrates theming according to a preferred embodiment.

Figure 18 schematically illustrates defined areas of an input surface of an embedded device for character recognition according to a preferred embodiment.

Figure 19 schematically further illustrates defined areas of an input surface of an embedded device for character recognition according to a preferred embodiment.

Figure 20 schematically illustrates different contention locking levels.

Figure 21 schematically illustrates relationships between different contention locking levels.

Figure 22 schematically illustrates a contention locking scheme according to a preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A preferred overall software architecture for use with an embedded device includes client software 2 and server software 4, and is schematically illustrated at Figure 1. It may be subdivided into four different layers, including three layers (1-3 below) and a server layer (4 below):

1. Operating System (6)
2. Programming Environment (8)
3. Application Framework (10)
4. Client Support Services (4)

The preferred architecture is based on a distributed computing model and uses a dedicated server component, or client support server 4 to offload computing tasks from the client 2, and to provide on-demand installable SW components. Each of the mentioned layers provides certain advantageous features, as described herein, in accordance with what is desired in the field of embedded device computing, e.g., as set forth above.

The operating system (OS) layer 6 includes an abstraction layer 14 which enables the use of a standard operating system 16 and usage of third party components like device drivers 18, and also provides the flexibility to exchange these operating systems 18 without affecting the rest of the system. This features relates particularly to features described below with respect to graphics rendering mechanisms on small embedded devices.

The programming environment layer 8 is preferably implemented as a Java virtual machine and corresponding libraries. This facilitates several features which permit enhanced performance (locking mechanism, see below), provides advantageous input methods (handwriting recognition, see below), and enables configurable user interfaces (theming 20, see below). The application layer has a query system module 22 which communicates with a unified data storage module 24. The unified data storage module 24, in turn, communicates with the programming environment 8 and participates in the theming 20.

The application framework 10 includes an advantageous application model, which provides an automated, on-demand triggered package management system via a package manager module which includes a package registry 28. This mechanism is preferred for enabling on-demand updates/installation of new software components via the network, and is described in more detail below. An application 29 is also shown running within the application framework 10 of the client 2.

The client support server 4 provides the backend for this package management and in this sense includes a package repository module 30, as shown. In addition, the client support server 4 has functions to transform complex content and/or protocols into simpler ones, having a type converter module 32 for translating content 36 received from a content server 38 and a protocol converter module 34 for mapping protocols from a repository 40 on a content server 38, thus offloading expensive

functions from the client 2. The client 2 may have capability to directly receive some content 42 from a content server 38 or otherwise such as in a peer-to-peer networking environment, wherein it is understood that when “content server” is used herein, it is meant to include any device from which content may be obtained.

CLIENT SUPPORT SERVICES

As mentioned above, embedded devices are typically memory constraint (i.e., have a limited amount of RAM/ROM), and typically do not have secondary storage (disks). Embedded devices also differ significantly from desktop computers in terms of processing power, screen resolution, and visuals (color-model). As a consequence, it is more difficult to install all potentially interesting software components before shipment of the device and/or to run all applications on all embedded device types, e.g., PDAs, cell phones, etc. A preferred embedded device does, however, include some network connection capability. Therefore, it is a feature of the preferred software architecture to provide reliable and efficient access to software and applications by the embedded device through network connection to the client support server 4 introduced above with reference to Figure 1.

The application model 10 of the client software 2 makes it possible to offload certain tasks to, and to automatically install software components from, a dedicated server 12, providing so-called “client support services” to the client 2. While the content server 38 would typically not be “client-aware” (i.e., with respect to architecture, capabilities, installed software, etc.), the client support server 38 would preferably be client-aware. The client application model / framework 10 is preferably designed in a way that it automatically reaches out to a well known (configured) client

support server 4 in case it encounters some content from the content server 38 that the client 2 does not know about, or that the client 2 cannot handle locally.

Referring now to Figure 2, the client 2 may retrieve certain types of content, e.g., content type A, directly from the content server 38 (or a generic proxy of it), using data path (A). In case this is not possible because of client constraints, either path (B) via the content type converter 32 or path (C) via protocol converter 34 are available through the client support services server 4. Path (D) is described in more detail below.

Referring now to Figure 3, a communication may be, although not necessarily, initiated by encountering a URI or content data (36 or 40, e.g.) which the client 2 is not configured to process, as illustrated by step (1). The client 2 then communicates a conversion request to the client support server 4 as illustrated by step (2). If the complex content data was supplied with the request, then the process would proceed to step (4). However, in case the complex content data was not supplied with the request, then the client support server 4 retrieves it from the content server as illustrated at step (3). In either case, the client support server converts the data, e.g., by translating complex content 36 to simplified content using the type converter module 32 or by mapping a complex protocol 40 to a simplified protocol using the protocol converter 34, each as illustrated at Figures 1 and 2. Finally, the client support server 4 sends the converted content back to the client 2 in a format/protocol that the client 2 can process as illustrated at step (4).

A typical format conversion example could be a complex word-processor document, wherein the embedded device, such as a handheld device, is configured such that it is only capable of displaying HTML or other markup language type content. In this example, the type converter 32 would convert the word processor document to markup language type. A protocol conversion example could be

accessing a mail server via a simple message protocol, leaving the complexities to handle IMAP, POP3, etc., to the protocol converter service 34 running on the client support server 4.

The client update/extension mechanism (D) (see Figure 2) will now be described with reference to Figure 4. The client update/extension mechanism (D) may typically start with a content reference (URI) from the content server 38 that the client 2 does not know about or is not configured with, as having only application 1 (29) installed, as illustrated by step (1) in Figure 4. For example, the client 2 may have no registered protocol-handler to retrieve the content, or may have no registered viewer for its format. In this case, a feature request is sent to the package repository service 30 on the client support server 4 as illustrated by step (2). The feature is looked up in the repository 30 (containing client software packages), and sent back as a package description (metadata) as illustrated by step (3), containing dependencies and conflicts to/with other packages. This is checked on the client against installed packages, potentially resulting in additional requests for missing dependency packages, i.e., repeating step (2).

Once all dependencies have been resolved, the required package data is retrieved from the client support server 4 as step (5) and finally processed as an install transaction (i.e., fully processed or reverted) of application 1 (39) at step (6). The user of the embedded device may be involved in the process by initiating a confirming request at step (4), although the system may be configured to automatically perform the process. In either case, the user doesn't not have to know about the feature with respect to package mapping, dependencies or constraints. Alternatively, step (3) may involve a user selection from a list of alternative packages.

Preferably, all client support services are utilized via an application model which is driven by portable documents written in a markup language (e.g., HTML or

XML). Documents can be nested, which enables a local service to embed a new child document into its requester. New local services may be installed via the package manager 28, and may in turn use a remote converter service, as described below with reference to Figure 5.

Referring to Figure 5, an illustration of how the process is performed by the application 29, the application framework 10 and the client support services server 4 is described below. The application 29 performs the steps of retrieving the document 42, parsing the document 44 and resolving features 46. The application framework 10 then becomes involved for package management 26, and communicates with the package repository 30 of the client support server 4. The package manager 26 communicates with a local service registry 48 which utilizes a local service 50. The local service communicates with the converter service 32, 34 of the client support server 4. The application 29 processes data via registered services 48 by communicating with the local service registry 48 and the local service 50. The document is cached 54, wherein the process may return to step 52. Resources are freed by the application at step 56.

GRAPHICS RENDERING MECHANISM

A graphics rendering mechanism for use with an embedded device which is designed to compensate for a lack of expensive display hardware is described below. The preferred graphics rendering mechanism for use with an embedded computing device includes an application 58, e.g, paint 59, a graphics toolkit 60, e.g, including drawling and fill rectangle tools 62, and a graphics driver 64. The graphics driver 64 is divided into two parts: a framebuffer access macro layer 66 and a shape function layer 68. The framebuffer access macro layer 66 includes a target architecture (CPU)

specific instruction set to set/retrieve pixel values into/from the framebuffer memory. The framebuffer access instructions are expanded or inlined into the shape function layer 68, to avoid or reduce call overhead. The shape function layer 68 itself can be inlined into the application, thus providing enhanced efficiency (at the cost of runtime configurability, which is less relevant for embedded systems as compared with, e.g., desktop systems).

One of the most performance critical tasks for a graphics driver is to map certain mathematically described shapes (lines, circles etc) to the display pixel raster, which involves finding the closest raster pixel for a given mathematical point. This is called rasterization.

The preferred rasterizer utilizes the mentioned characteristics of embedded devices, to compensate lack of expensive display hardware by means of

- aggressive inlining
- processor specific acceleration of scanline access functions

Display adapter acceleration can still be done at the shape function level 68 (e.g. drawLine), but the preferred mechanism introduces a whole new layer of processor specific acceleration, which is implemented as a set of macros, i.e., enables inlining up to a point where there is little or no indirect invocation of rendering functions.

Most shape algorithms are conceptually 2-dimensional, i.e., assume a logical framebuffer model which can be thought of as a matrix of pixels, each addressed by a x,y coordinate pair. However, the physical framebuffer is just a simple memory range, i.e., 1-dimensional, organized into so called scanlines (the matrix rows). The preferred rasterizer bridges this gap by means of the fact that most shape function algorithms make use of shape-specific scene cohesion properties, where new pixel positions are derived from previous pixel positions. The Bresenham line drawing

algorithm can be considered as a typical example: depending on the slope of the line, the algorithm iterates in 1-increments through the major delta coordinate (e.g., x), and accumulates error terms for the dependent pixel coordinate (e.g., y). If the error term (which is usually inexpensive to evaluate) exceeds a certain threshold, the dependent pixel coordinate is incremented (see Figure 7).

Such an algorithm can be re-formulated so that it uses an abstract set of scanline access instructions:

- `get_SLC(x,y) -> s`

compute linear pixel address for a given x,y point

- `set_SLC(s,color)`

set the color value of a given pixel address

- `set_SLC_rows(s,len,color)`

set color values for a consecutive range of pixels

- `copy_SLC_rows(s,len,d)`

copy color values for consecutive range of pixels from other location

- `inc_SLC(s) -> s`

increment the pixel address by 1

- `add_SLC(s,n) -> s`

increment the pixel address by a given number

These instructions can be implemented as processor specific macros, which advantageously can have a significant performance gain for setting/copying pixel ranges (i.e., block transfer operations used in filling rectangles, drawing horizontal/vertical lines, copying bitmaps, etc.). Compared to a generic `setPixel(x,y,color)` function, this also eliminates a significant number of expensive multiplication instructions, replacing them by simple additions, or even increments.

By means of these pseudo instructions, the 2-dimensional shape construction is separated from configurable 1-dimensional, processor specific scanline functions.

This separation is based on the concept of the SLC, the build-time configured scanline cell type, which is the smallest addressable scanline unit holding the color information, e.g., for at least one pixel (see Figures 8 and 9). This abstraction is used in order to deal with visuals which pack several pixels into one byte (e.g. black/white, or VGA). These visuals can be dealt with by means of adding a column parameter (x) to the mentioned scanline access instruction set.

FONT EXTENSIONS

To overcome the problem set forth above with respect to the unicode set and the limited memory capacity available on embedded devices, as illustrated at Figure 10, a preferred font handling mechanism is now described with reference to Figure 11. The preferred font-handling mechanism is based on the assumption that for most localizations, only a single primary character subset is used from the whole unicode set. There might be characters from additional, secondary subsets interspersed, but these sets are usually smaller, and are just used for mixed in words/names. As a example, a typical Japanese localization uses about 8000 characters from the Kanji/Katakana/Hiragana unicode subsets, with occasional western names rendered in latin characters. Such a font instance would typically add up to about 500kB.

The preferred font handling mechanism therefore assumes that (especially per-character) font attributes are kept in disjunct, but linked memory objects, so called *font-extensions*. In case of a very expensive extension (e.g., Kanji), this extension could be shared between several font instances, especially for different per-font rendering attributes like *italic*, which mostly make sense for latin font sets anyway. In

addition, font-extensions can be loaded on demand, which is particularly advantageous on systems not providing memory mapped font instances (i.e., filesystem data is duplicated into the process address space).

By splitting up the whole unicode font range into configured primary/secondary font extensions (with possibly shared secondary extension data), it becomes feasible to implement a reasonable number of font instances in less than 2MB of memory. The fact that most of the text is rendered in the primary character subset, and secondary subsets are used on a per-word basis (i.e. the number of extension switches is relatively low), can be used to compensate the efficiency penalties (caused by per-character font-extension checking) by means of keeping a reference to the most-recently-used extension.

CLIPPING

A structure and algorithm is now described which compensates for the lack of embedded devices having a sophisticated display processor, and enables a straightforward, inexpensive implementation of the rasterizer, which is preferably based on rectangular clipping at the driver level. As mentioned above, the preferred embedded device is configured to display drawings surfaces or windows overlapping one another on the display.

Figure 12 describes a preferred algorithm by means of a UML-like diagram, with boxes representing types/classes (including name, data members and functions), round boxes representing snippets of the key functions, thin arrow lines representing references, thick arrow lines representing inheritance. A first feature of the preferred algorithm involves an object which is of type 'DrawingSurface', and it denotes a UI component (i.e. Window) that has a z-

order attribute (stacking order), i.e., can be obscured by siblings. Another feature involves a second object which is of type 'GraphicsContext', which bundles rendering attributes like font and colors. A 'GraphicsContext' instance is preferably attached to a single 'DrawingSurface' object, i.e., includes a reference to it which is set during the 'GraphicsContext' initialization.

One of the rendering attributes of a 'GraphicsContext' object can be an explicitly set clipping region ('clipRegion'), i.e., the area to which graphics output is confined to (which is application specific). Graphics context objects can have explicitly set attributes like color, font, and clipping regions. When an output operation is performed, a visibility tag (see below) of the attached surface object is checked for changes. If the tag value has been modified, a set of rectangular clip segments are computed and stored in the graphics context object, by means of calculating the intersection of the drawing surface's visible segments with the graphics context's clipping region:

$$\{R_{clipSeg}\}_{context} = \{R_{visSeg}\}_{surface} \text{ intersected with } \{R_{clip}\}_{context}$$

Additional advantageous features of the preferred algorithm include "visible segments", "clip segments", and "visibility tags". Visible segments ('visSegments'), are sets of rectangles describing the visible, not obscured region of a 'DrawingSurface'. These visible segments are preferably changeable by modifying the z-order of the 'DrawingSurface' object (e.g., by calling a function 'changeZorder()'), which is assumed to happen orders of magnitudes less frequent than executing the drawing operations using these segments (e.g. 'drawShape()'), and are kept as attributes of the 'DrawingSurface'. Every change of the visible segment set increments a visibility tag ('visTag'), which is also stored as a 'DrawingSurface' attribute.

Clip segments ('clipSegments'), describe the exposed region of a 'DrawingSurface' instance which falls within the clipping region of a

'GraphicsContext' instance. When an output operation (e.g. 'drawShape()') is executed via a 'GraphicsContext' object, the visibility tag stored in the 'GraphicsContext' object is compared to the one stored in the corresponding 'DrawingSurface'. If the tag value has been modified, a new set of rectangular clip segments is computed and stored in the 'GraphicsContext' object, by means of calculating the intersection of the visible segments of the 'DrawingSurface' with the clipping region of the 'GraphicsContext'.

The output operation is then performed iterative for each clip segment rectangle, i.e., preferably does not use more than simple rectangular clipping from the underlying graphics driver / rasterizer.

:

```
GraphicsContext::drawShape (..) ::=
..
foreach ( rect in clipSegments )
    displayDriver.drawShape( rect)
..
```

Referring to Figure 13, the computation of visible segments and clip segments is done by means of an intersection algorithm which treats rectangles and sets of non-overlapping rectangles (aka regions) uniformly, the underlying data model being a linkable rectangle (i.e., sets represented as a linked list of rectangles). For each combination of two overlapping rectangles R_1 and R_2 , the inverse logical product is either nil (R_1 fully obscured by R_2), a single rectangle (R_1 left/upper/right/lower half obscured by R_2), or a set of 2 to 4 rectangles, as illustrated at Figure 14.

Calculating intersections of sets of rectangles $\{R_1\}$ and $\{R_2\}$ is performed by building the union of result sets obtained by computing the inverse logical product for

each combination of elements of $\{R_1\}$ and $\{R_2\}$. The preferred clipping mechanism is especially useful in the context of non-overlapping drawing surfaces (i.e., a small number of resulting clip segments, which are infrequently modified), since it includes only minimal overhead to handle set operations, and avoids expensive clip region re-calculations.

THEMING

The following embodiment is particularly contemplated for use with embedded devices wherein a vendor and/or user configured look and feel is desired (e.g., colors, fonts, decorations), which is referred to herein as theming. To achieve this goal, a preferred software scheme includes a design which makes it possible to generically separate the logic of a user interface component from data which can be used to factorize the way it is rendered.

Referring to Figure 15, the design is based on an abstraction that each UI component can be divided into a border and a background area, the border having invariant extensions for a specific component class (e.g. Button) and theme. Information which is displayed in the UI component (e.g., a Button label text) is drawn on top of the background, is clipped against the border, and is rendered in a separate step.

Rendering of the border and background may be performed automatically by the system (i.e., does not involve user code), and uses a per-UI component decoration object which may be theme-, UI class-, and style- specific. The decoration object can be shared between instances of the same UI class, and gets set during the initialization of the UI via a configured theme object, which is a decoration factory.

This process is described in Figure 16, by means of a UML-like diagram,

with boxes representing types/classes (including name, data members and functions), round boxes representing snippets of the key functions, thin arrow lines representing references, thick arrow lines representing inheritance relationships (arrow pointing towards super-class), and dotted arrow lines representing object creation. Attributes which are used to render UI data (colors, font, clipping region), and to determine the UI component size (border widths), are queried from the decoration object. The preferred mechanism enables very flexible rendering schemes, e.g., partially transparent borders, backgrounds inherited by UI component-parents, and UI-state/-style specific representation (e.g., 'focused' UI components). Theme and decoration classes can be completely hidden from user code, which makes the mechanism especially suitable for extending the functionality of existing application programming interfaces, i.e., to use theming in applications which are compatible to standard libraries (like Java). In addition to this generic design, the preferred configuration also includes a specialized ConcreteDecoration class as illustrated in Figure 16, which is based on a tiled image model.

There are three major types of objects involved in the preferred embodiment: "user interface components" (also called "widgets"), "decoration" objects, and "theme" objects, each of them derived from abstract base types, with at least one concrete derived type (e.g. 'AbstractWidget' / 'ConcreteWidget'). UI component objects are the objects used in the application (e.g., a button), to display data (e.g. text), and to collect user input. Decoration objects are system constructs to separate the UI component logic from rendering border and background of a UI component. Theme objects are used to create specific decoration objects so that their types do not have to be hardcoded into the UI components.

The common UI component type ('AbstractWidget') references a 'AbstractDecoration' object, which is obtained from a global theme object ('AbstractTheme') acting as a abstract factory pattern (creating specific

'ConcreteDecoration' instances and returning them as 'AbstractDecoration' references). Creating and storing decoration objects is usually done as part of the UI component initialization (e.g. in `initialize()`). Once the 'decoration' object is known to the UI component, it uses it to obtain its own dimensions (e.g. by calling `'getBorderWidth()'`), which is done by adding the decoration-specific border extensions to its data-dependent background extensions.

The rendering process itself is divided into two separate layers: (1) automatic decoration rendering and (2) component specific data rendering. It is wrapped into a function which is automatically called by the system (e.g. `'render()'`), implemented in the common UI component type (e.g., `'AbstractWidget'`), which first uses the decoration object to render border and background areas (e.g. by calling `'drawBorder()'`, `'drawBackground()'`), and then calls a overloaded rendering function (e.g. `'paint()'`), which is implemented in the concrete UI component class to render the component specific data on top of the background area.

Referring now to Figure 17, there are preferably nine images (bitmaps) used per widget, one for each corner, one for each border, and one for the background. Border images are tiled left-to-right, top-to-bottom, and the background image is tiled across the whole interior of the widget.

This decoration scheme is especially advantageous for implementing artistic themes, i.e., providing customized look and feel without programming. The theme images can be kept in separate packages/archives which can be installed separately.

According to this preferred embodiment, a rendering policy is encapsulated into a shared delegation object (the 'decoration'), i.e., an API (not just data). A UI component area is logically split up into background and border. The border dimensions are queried and attributes rendered (e.g., font,color) from the decoration object, and the UI component size is calculated from it. The drawing is preferably

split up into a non-overridable system part (e.g., "render()" in the picture) and an overridable user part (e.g., "paint()").

HANDWRITING RECOGNITION

This following description relates to a preferred handwriting recognition system which is designed to provide the following advantages particularly for use with embedded devices:

- fast and small so enabled for use on low performance processors and with low memory footprints
- Uses integer arithmetic since most embedded processors do not have floating point ability
- Recognizes uni-stroke characters such as may be used on Palm and WinCE devices, such that it can recognize both Grafitti (Palm) and JOT (WinCE + others)
- extensible to support multi-stroke characters such as non-latin language support

The preferred algorithm presented here preferably includes the following features:

1. Character strokes are encoded as an integer value
2. A small number multiple (as small as one) integer encoding represents each recognizable character
3. Strokes are encoded by observing the way characters are drawn rather than by what the character actually looks like
4. The character encoding includes a series of direction events denoting changes in movements north, east, south and west

5. The character encoding can be extended to improve recognition by adding special events to note specific actions. These actions include termination of a stroke near the beginning point (allowing better recognition of characters such as 'O', 'B', etc.), as well as pen up-down events to allow for encoding of multi-stroke characters.

The following algorithm is preferably used wherein stroke characters are converted into integers:

Step A: Setup

1. Five event values are defined as follows:

north = 0, east = 1, south = 2, west = 3, closed = 4.

Since these values can be represented in 3 bits each, a shift value is also defined, e.g., shift = 3.

2. The sample distance is defined. This value is the minimum distance between position samples for a new sample to be processed by the conversion algorithm.

dest = <some value>

3. The end distance is defined. This value is the maximum distance between the pen down position and the pen up position for a stroke to be considered 'closed'. An example of a closed stroke is the letter 'O' which starts and ends in the same place.

edest = <some value>

Step B. Pen Down

1. When the pen is pressed we record the initial x,y coordinates in a number of places.

- a. firstx,firsty - are used to remember the start position of the stroke.
 - b. lastx,lasty - are used to remember the last x and y values when we generated a new event value
 - c. leftx, rightx - are defined with the initial x value. These are later used to determine which character set to use during final recognition
2. We also reset the north-south and east-west recognition state
north-south = unknown, east-west = unknown
 3. Set the pattern to zero

Step C. Pen Move

1. A large number of pen move events are generated as the pen moves. Each point is analysed as follows
2. if the x position is less than the current value of leftx then make leftx equals x
3. if the x position is greater than the current value of rightx then make rightx equals x
4. Determine change in north/south motion:
5. If y has moved by more than +dist then we are moving north so set newdir to be north
6. Else if y has moved by more than -dist then we are moving south so set newdir to be south
7. Else the change in y is too small to be interesting. Jump to analyse the x value (step 13)

8. If the value of newdir is the same as the value of north-south then we're moving in the same direction as before and don't need to do anything else. Jump to analyse the x value (step 13)
9. If north-south isn't set to unknown set the value of east-west to unknown. This forces the generation of a new east-west event whenever we change our north-south direction
10. Set north-south to the new direction newdir so we don't generate this direction again
11. Set the lasty value to y
12. Add to the recognized pattern as follows
pattern = pattern leftshifted-by shift
pattern = pattern bitwise-or newdir
13. Determine change in east/west motion:
14. If x has moved by more than +dist then we are moving east so set newdir to be east
15. Else if x has move by more than -dist then we are moving west so set newdir to be west
16. Else the change in x is too small to be interesting. Jump to the end (step 22)
17. If the value of newdir is the same as the value of east-west then we're moving in the same direction as before and don't need to do anything else. Jump to then end (step 22)
18. If east-west isn't set to unknown set the value of north-south to unknown. This forces the generation of a new north-south event whenever we change our east-west direction

19. Set east-west to the new direction newdir so we don't generate this direction again

20. Set the lastx value to x

21. Add to the recognized pattern as follows

pattern = pattern leftshifted-by shift

pattern = pattern bitwise-or newdir

22. Return. We have filtered and generated a new pattern based on this movement. We keep building the pattern as long as we move the pen

Step D. Pen Up

1. If the distance between x and firstx is less than edist and the distance between y and firsty is also less than edist then we have finished our stroke where we started it. In this case we add a 'closed' event to the pattern as follows

pattern = pattern leftshifted-by shift

pattern = pattern bitwise-or closed

2. For uni-stroke characters we have finished generating a character pattern and must now convert it into a character

3. The writing area looks like the illustration shown at Figure 18 and is preferably based on the JOT system. It is determined which part of the area was written in to determine which character set to use for recognition.

4. If the leftx value is less than 'A' then we wrote in the SYM area.
Set charset to SYM

5. If the leftx value is greater than 'B' then we wrote in the NUM area. Set charset to NUM

6. If the leftx value is less than 'B' and the rightx value is greater than B then we wrote of the 'CAP' line. Set charset to CAP
7. Else set charset to CHAR
8. Within the selected charset look for an exact match for the final pattern value. If one is found return the associated character
9. Otherwise no match is found.

The above algorithm can be extended to recognize multi-stroke characters. To do this, the event 'pen down' is defined which is added to the pattern in the usual way whenever the pen is pressed down. However, to avoid recognition when the pen is released, 'Step D' is triggered on another action rather than 'pen up' as defined for uni-stroke characters. This action can be one of the following:

1. Timeout - the pen can go down and up as many times as required but if it up for more than X milliseconds we consider the character complete and trigger 'Step D'
2. Progressive timeout - when the pen is released, the character is recognized. If recognized, then the character is complete and the algorithm moves onto the next character. If the recognition fails, the addition of new strokes to the pattern are allowed. If the pen is released for X milliseconds and a character is not yet recognized, the algorithm is reset and restarted.

Spatial - an area may be defined as illustrated in Figure 19. A character is written in each box from left to right (or right to left depending on the language). A character is considered complete and trigger 'step D' when the pen is put down in a different box from the last box. Note that this scheme does not allow the separate recognition of symbols, numbers, uppercase letters and lowercase letters. However, all of these may be encoded using multiple strokes making such a scheme unnecessary.

LOCKING MECHANISM

In order to provide a safe multi-threading environment for applications (concurrent execution of several tasks), the preferred embodiment includes an efficient inter-thread synchronization scheme, which is done by means of monitors (guarded sequences of instructions, allowing just a single thread at a time to execute the instructions), and using efficient object locks. Depending on the thread model in use, checking, and updating object locks can be very expensive, since these operations involve atomic instruction sequences which might be implemented as operating system-specific system functions. Advantageously, the locking mechanism according to the preferred embodiment provides testing and setting of the lock in an uninterrupted manner:

```
if object not locked
    /* no context switch allowed here */
    lock object by current thread
```

The scheduler (or thread switching mechanism) is not allowed to switch threads between the lock test and the lock set, which avoids inconsistencies (e.g., object lock held by two different threads). This is achieved at the scheduler level, i.e., is implemented efficiently with user thread systems (wherein the scheduler is part of the application), but can use expensive system calls in case kernel threads are used.

While the probability of colliding lock attempts depends on the nature of executed applications, statistical analysis has shown that in typical applications this probability is less than 0.1, i.e. more than 90% of all locking operations release the lock before another thread tries to get it. However, it is quite common that a lock is recursively obtained/released from within the same thread, as in:

```
function foo
    lock A
```

call bar ---> function bar
lock A

This does not block execution in function bar, since the lock at this point already has been acquired by the current thread.

The well-separated nature of locking attempts is depicted at Figure 20. The non-overlapping locks and recursive thread internal locks are deemed un-critical in that not more than one thread tries to obtain the lock at a given time. The lock collision area is the only case which involves a second requestor (thread B) which is blocked on the attempt to acquire the lock, to be re-scheduled once the lock is given up (by thread A).

The locking aspect of this preferred embodiment builds upon this observation to distinguish between heavy and light locks, handling the much more likely light locks (non-overlapping and recursive thread internal locks) in user space, i.e., without the use of expensive system functions, and reverting to system-based heavy locks (lock collision situations) when a potential collision is not recognized as just a thread-internal nested locking attempt.

Each object can have one of the lock states depicted in Figure 21. The first locking attempt puts an unlocked object into the 'light locked' state. If there are subsequent locking attempts from within the same thread, the state doesn't not change. However, if another thread tries to lock a 'light locked' object, the state changes to 'heavy locked'.

In order to implement this model, the preferred embodiment uses three advantageous constructs

- lock structures (containing the lock holder and wait queues)
- stack-based *lock slots* (local lock structure variable)
- a lock structure reference in each object header

Typically, nested intra-thread lock attempts are detected by means of obtaining an identifier of the currently running thread, and to compare it with the one stored in the lock structure. However, obtaining such a thread id can be expensive in itself (especially for kernel thread systems, where it again involves a system call, i.e., requires the measure that the preferred embodiment overcomes by means of the light locks). The preferred embodiment works around this by using addresses of stack variables to identify threads (stacks are per-thread data areas). If a locking attempt of a 'light locked' object detects that the difference $D = S_1 - S_2$ between its own lock slot and the address stored in the object lock reference is smaller than a certain threshold (deduced from a reserved area at the end of each stack), this is recognized as a nested intra-thread locking attempt and the object lock is not inflated (i.e., it is not turned into a heavy lock). The feature is illustrated at Figure 22. While this conservative approach might inflate some locks without need (inside of deeply nested function calls), the vast majority of nested locking attempts is recognized and handled without overhead.

To provide atomic execution of the lock test & set, the preferred embodiment uses a generalized compareAndExchange macro, which is logically defined by

```
compareAndExchange ( address, compareValue, newValue ) ::=
  if value at address equals compareValue
    set value at address to newValue
    → true
  else
    → false
```

To achieve portability, the preferred embodiment does not assume that the target architecture (CPU) directly supports the compareAndExchange function, and provides implementations for several different atomic machine instructions:

1. COMPARE_AND_EXCHANGE (e.g. cmpxchgl on x86)
2. ATOMIC_EXCHANGE (e.g. swp on ARM)

3. TEST_AND_SET (e.g. tas on SHx)

While implementation with (1) is most straight forward, (2) and (3) involve the use of additional helper variables.

ATOMIC_EXCHANGE uses a stack variable to store the results of the atomic instruction in a thread-safe location

```
COMPARE_AND_EXCHANGE( Lock** a, Lock* o, Lock* n) ::=
    Lock* val = LOCKINPROGRESS;
    ATOMIC_EXCHANGE(a, val);

    if (val == o)
        *a = n; → true
    else if (val == LOCKINPROGRESS)
        → false
    else
        *a = o; → false
```

The temporary LOCKINPROGRESS value is used to detect concurrent executions of COMPARE_AND_EXCHANGE.

TEST_AND_SET can be used to efficiently implement a global lock

```
COMPARE_AND_EXCHANGE(Lock** a, Lock* o, Lock* n) ::=
    static int taslock;

    while (TEST_AND_SET(&taslock) == 0);

    if (*a == o)
        *a = n; taslock = 0; → true
    else
        taslock = 0; → false
```

To distinguish between light and heavy locks, the invention makes use of word aligned lock slots and heavy lock objects, setting bit 0 when storing heavy lock addresses in the object lock field.

```
heavyLock(F,adr) ::=
    ..
```

```
lock *lkp = getLockObject(..)
*F = (lkp | 1)
```

The IS_HEAVY test macro then simply has to check for this bit

```
IS_HEAVY(F) ::=
if ( (*F & 1) != 0 )
    → true
else
    → false
```

With these functions, the lock and unlock operations can be implemented like

```
lock( Lock** lkp, void* where) ::=
    uintp val = (uintp)*lkp;

    if (val == 0)
        if (!COMPARE_AND_EXCHANGE(lkp, 0, (Lock*)where))
            heavyLock(lkp, where);
        else if ((val - (uintp)where) > (STACKREDZONE / 2) )
            heavyLock(lkp, where);
        else /* recursive intra-thread lock */
and
    unlock(Lock** lkp, void* where) ::=
        uintp val = (uintp)*lkp;

        if (IS_HEAVY(val))
            heavyUnlock(lkp, where);
        else if (val == (uintp)where &&
!COMPARE_AND_EXCHANGE(lkp, (iLock*)where, LOCKFREE))
            heavyUnlock(lkp, where);
```

In summary, the preferred embodiment uses stack variables, memory alignment (of stack variables and heap objects), and an abstract COMPARE_AND_EXCHANGE instruction to achieve a portable, thread-system neutral, and efficient contention-based lightweight locking scheme.

HTML CORRECTION

The application model according to a preferred embodiment is based on markup language documents, the most common language being HTML.

Unfortunately, this language has evolved over more than a decade, with the result that many documents today do not follow a strict grammar, or are simply malformed.

Classical examples of malformed HTML are missing end-tags

```
<UL>
  <LI> one
  <LI> two
</UL>
or overlapping tags
```

```
<FONT ..>
  <TD> one
</FONT>
</TD>
```

Constructing meaningful documents from such input usually involves a separate parsing step, i.e., generating a memory representation, correcting it, and then writing it back so that it can be further processed/displayed (which again involves a parsing step, with different semantic actions and node types).

The preferred embodiment works around this problem by means of utilizing a recursive descend parsing scheme with token lookahead (mapping grammar productions into recursively called functions), which enables to return from productions based on semantic actions. The following grammar fragment prints semantic actions in bold italics, and uses a EBNF like syntax definition ('{..}' denoting repetitions, '[..]' denoting options, and '|' separating choices).

```
node ::= '<' START_TAG '>'
      {   if (!validNextNode(START_TAG,
          lookAhead(1),lookAhead(2))) return; (1)
        node [TEXT]
      | '<' END_TAG '>' if (equal(START_TAG,END_TAG)) return; (2)
      }
```

The semantic action (1) is executed before the parser descends into the next 'node' construct, which would be treated as a child of START_TAG.

The preferred embodiment introduces a function `validNextNode()`, which is responsible for determining if the following node is either (A) a valid child node or (B) an acceptable end node of `START_TAG`.

Test (A) can be efficiently implemented by means of a $N \times N$ matrix of all valid HTML tags, with '1' elements indicating a valid parent-child relationship. This test would fail for the `..` combination of the first example, bailing out of the first ``, then accepting the second `` as a valid child of the enclosing ``.

Test (B) requires the classification of tags as being 'structural' (like ``), or 'attribute' (like ``). Mismatching structural end tags are preferably not accepted, and are returned to their corresponding `START_TAG` level. Mis-matching attribute tags preferably are usually accepted, since they are mainly used for semantic side effects (like setting attribute values for all enclosed "child" nodes). A matching end tag of course returns after consuming all corresponding end tokens (2). With this mechanism, it becomes possible to parse and process HTML documents (showing the above deficiencies) in a single pass.

While exemplary drawings and specific embodiments of the present invention have been described and illustrated, it is to be understood that that the scope of the present invention is not to be limited to the particular embodiments discussed. Thus, the embodiments shall be regarded as illustrative rather than restrictive, and it should be understood that variations may be made in those embodiments by workers skilled in the arts without departing from the scope of the present invention as set forth in the claims that follow, and equivalents thereof.

In addition, in the method claims that follow, the operations have been ordered in selected typographical sequences. However, the sequences have been selected and so ordered for typographical convenience and are not intended to imply any particular order for performing the operations, except for those claims wherein a particular

ordering of steps is expressly set forth or understood by one of ordinary skill in the art as being necessary.

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 2517 2518 2519 2520 2521 2522 2523 2524 2525 2526 2527 2528 2529 2530 2531 2532 2533 2534 2535 2536 2537 2538 2539 2540 2541 2542 2543 2544 2545 2546 2547 2548 2549 2550 2551 2552 2553 2554 2555 2556 2557 2558 2559 2560 2561 2562 2563 2564 2565 2566 2567 2568 2569 2570 2571 2572 2573 2574 2575 2576 2577 2578 2579 2580 2581 2582 2583 2584 2585 2586 2587 2588 2589 2590 2591 2592 2593 2594 2595 2596 2597 2598 2599 2600 2601 2602 2603 2604 2605 2606 2607 2608 2609 2610 2611 2612 2613 2614 2615 2616 2617 2618 2619 2620 2621 2622 2623 2624 2625 2626 2627 2628 2629 2630 2631 2632 2633 2634 2635 2636 2637 2638 2639 2640 2641 2642 2643 2644 2645 2646 2647 2648 2649 2650 2651 2652 2653 2654 2655 2656 2657 2658 2659 2660 2661 2662 2663 2664 2665 2666 2667 2668 2669 2670 2671 2672 2673 2674 2675 2676 2677 2678 2679 2680 2681 2682 2683 2684 2685 2686 2687 2688 2689 2690 2691 2692 2693 2694 2695 2696 2697 2698 2699 2700 2701 2702 2703 2704 2705 2706 2707 2708 2709 2710 2711 2712 2713 2714 2715 2716 2717 2718 2719 2720 2721 2722 2723 2724 2725 2726 2727 2728 2729 2730 2731 2732 2733 2734 2735 2736 2737 2738 2739 2740 2741 2742 2743 2744 2745 2746 2747 2748 2749 2750 2751 2752 2753 2754 2755 2756 2757 2758 2759 2760 2761 2762 2763 2764 2765 2766 2767 2768 2769 2770 2771 2772 2773 2774 2775 2776 2777 2778 2779 2780 2781 2782 2783 2784 2785 2786 2787 2788 2789 2790 2791 2792 2793 2794 2795 2796 2797 2798 2799 2800 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845 2846 2847 2848 2849 2850 2851 2852 2853 2854 2855 2856 2857 2858 2859 2860 2861 2862 2863 2864 2865 2866 2867 2868 2869 2870 2871 2872 2873 2874 2875 2876 2877 2878 2879 2880 2881 2882 2883 2884 2885 2886 2887 2888 2889 2890 2891 2892 2893 2894 2895 2896 2897 2898 2899 2900 2901 2902 2903 2904 2905 2906 2907 2908 2909 2910 2911 2912 2913 2914 2915 2916 2917 2918 2919 2920 2921 2922 2923 2924 2925 2926 2927 2928 2929 2930 2931 2932 2933 2934 2935 2936 2937 2938 2939 2940 2941 2942 2943 2944 2945 2946 2947 2948 2949 2950 2951 2952 2953 2954 2955 2956 2957 2958 2959 2960 2961 2962 2963 2964 2965 2966 2967 2968 2969 2970 2971 2972 2973 2974 2975 2976 2977 2978 2979 2980 2981 2982 2983 2984 2985 2986 2987 2988 2989 2990 2991 2992 2993 2994 2995 2996 2997 2998 2999 3000 3001 3002 3003 3004 3005 3006 3007 3008 3009 3010 3011 3012 3013 3014 3015 3016 3017 3018 3019 3020 3021 3022 3023 3024 3025 3026 3027 3028 3029 3030 3031 3032 3033 3034 3035 3036 3037 3038 3039 3040 3041 3042 3043 3044 3045 3046 3047 3048 3049 3050 3051 3052 3053 3054 3055 3056 3057 3058 3059 3060 3061 3062 3063 3064 3065 3066 3067 3068 3069 3070 3071 3072 3073 3074 3075 3076 3077 3078 3079 3080 3081 3082 3083 3084 3085 3086 3087 3088 3089 3090 3091 3092 3093 3094 3095 3096 3097 3098 3099 3100 3101 3102 3103 3104 3105 3106 3107 3108 3109 3110 3111 3112 3113 3114 3115 3116 3117 3118 3119 3120 3121 3122 3123 3124 3125 3126 3127 3128 3129 3130 3131 3132 3133 3134 3135 3136 3137 3138 3139 3140 3141 3142 3143 3144 3145 3146 3147 3148 3149 3150 3151 3152 3153 3154 3155 3156 3157 3158 3159 3160 3161 3162 3163 3164 3165 3166 3167 3168 3169 3170 3171 3172 3173 3174 3175 3176 3177 3178 3179 3180 3181 3182 3183 3184 3185 3186 3187 3188 3189 3190 3191 3192 3193 3194 3195 3196 3197 3198 3199 3200 3201 3202 3203 3204 3205 3206 3207 3208 3209 3210 3211 3212 3213 3214 3215 3216 3217 3218 3219 3220 3221 3222 3223 3224 3225 3226 3227 3228 3229 3230 3231 3232 3233 3234 3235 3236 3237 3238 3239 3240 3241 3242 3243 3244 3245 3246 3247 3248 3249 3250 3251 3252 3253 3254 3255 3256 3257 3258 3259 3260 3261 3262 3263 3264 3265 3266 3267 3268 3269 3270 3271 3272 3273 3274 3275 3276 3277 3278 3279 3280 3281 3282 3283 3284 3285 3286 3287 3288 3289 3290 3291 3292 3293 3294 3295 3296 3297 3298 3299 3300 3301 3302 3303 3304 3305 3306 3307 3308 3309 3310 3311 3312 3313 3314 3315 3316 3317 3318 3319 3320 3321 3322 3323 3324 3325 3326 3327 3328 3329 3330 3331 3332 3333 3334 3335 3336 3337 3338 3339 3340 3341 3342 3343 3344 3345 3346 3347 3348 3349 3350 3351 3352 3353 3354 3355 3356 3357 3358 3359 3360 3361 3362 3363 3364 3365 3366 3367 3368 3369 3370 3371 3372 3373 3374 3375 3376 3377 3378 3379 3380 3381 3382 3383 3384 3385 3386 3387 3388 3389 3390 3391 3392 3393 3394 3395 3396 3397 3398 3399 3400 3401 3402 3403 3404 3405 3406 3407 3408 3409 3410 3411 3412 3413 3414 3415 3416 3417 3418 3419 3420 3421 3422 3423 3424 3425 3426 3427 3428 3429 3430 3431 3432 3433 3434 3435 3436 3437 3438 3439 3440 3441 3442 3443 3444 3445 3446